A silver pickle fork

&

Some thoughts on metrics

"If I have seen further it is by standing on ye shoulders of Giants" -- Isaac Newton.

Note: 'ye' ='the' as the 'y' was the letter 'thorn' which represents a modern "th".

Improving the process of project management will not create as big an improvement in software system engineering, as would improving project management, standing on the top of modern software tools and techniques. (A giant standing on the shoulders of a giant).

An example of the use of a tool which aided in the creation of a new way of doing things, is Isaac Newton use of a 'high-lighter' on all his books, he invented a method of high-lighting the key points on each page by folding the page twice, in such a way as to have the folds cross at a keyword ("In the Presence of the Creator ISAAC NEWTON & HIS TIMES" by Gale E. Christianson). At a time when the book index had not been invented, having a free highlighter gave Isaac Newton an edge when it came to retrieving and recalling studied information.

The proceeding points are more directed at legacy systems (late in their life cycle software systems {such as COBOL business systems}) which are not being schedule to be re-engineered in the near future. These comments are being made to propose a way to help with the software crisis associated with maintaining current software systems during the last half of there life cycle. Not a silver bullet, but perhaps a silver pickle fork.

Nails were at one time so expensive that quite often the older structure being replaced were burned to the ground to retrieve the nails for new structure. Only later, after nails were mass-produced, did it make sense to conserve and standardize the wood used (creating a new wood product -- lumber), until that point any labor done on site with the wood was inexpensive compared to the cost of nails. <*add a reference to the building framing book* >.

In this example it was at one time clear that nails was the area of building that required the most attention. However after nails were mass-produced, then other areas could receive incremental improvement, which made building easier, while using less resources.

I will now propose some changes which may provide similar a productivity and quality improvement for (legacy) systems in the last half of there life cycle.

Some of the benefits of Object Oriented programming (information hiding and abstraction), can be achieved through modular programming which is within the abilities of almost all languages used in current legacy systems (including COBOL). Also code reuse can be facilitated through modular programming. Modules can be used to divide and then conquer the project by assigning to project team members appropriate modules (or the main program) to work on. Management would also have a metric of the progress being made on the system (If Configuration Management were used to check in all the modules that have been completed). Debugging and testing can be made easier through modularity (some of this due to being able to divide and conquer during the unit testing). Although I believe that most Legacy programmers have heard of modular programming, why has it been used to such a limited extent?

At one time “structured programming” was the method being pushed as the silver bullet for software quality and maintainability. If modularity would have been stressed instead, then structured programming would have naturally occurred in all small modules. Structured programming would almost occur naturally, when the program was designed as modules, which logically broke the program into parts that are related to each other.

I think if a survey of past (and perhaps present) COBOL college textbooks were to be taken, it would be found most of them do not teach how to use (or even mention) subroutines.

No legacy programmer would experiment with subroutines unless they were given time to. And they may fear criticism or worse, if they make a mistake using the “newer” technology. Even with the use of subroutines that does not imply that information hiding is being used and without subroutines all the variables are global in scope!

Legacy COBOL programs written for a IBM mainframe (or IBM mainframe compatible e.g. Amdahl) provide a facility for code reuse (for which I am sure training was provided) called “copylib”, which provide a primitive method for reusing code by coping paragraphs into each of the specified systems. This requires the same variable name to be used in the program as the reused paragraph, which encourages the use of global variables in COBOL program. A variable assignment from the variable used in the main program to the “global” variable used in the paragraph could be made, before the paragraph ‘call’; however no information hiding or Encapsulation would be involved.

Information hiding works somewhat like the Government’s security need to know principle. A module should only be passed the data it needs to have, to do its task, to do otherwise would increase the amount of data that could be changed by the module, and any change to data not needed by the module could only be an error. This is why global variables are bad, and should be used only in extreme circumstances (i.e. speed problems)!

A form of abstraction can be used during the design. Going from an abstract (big) picture of the forest, to an abstract picture of each grove of trees, to a less abstract picture of each clump of trees, then to each individual tree. The design of the software system can be abstracted in that way with the main program calling subroutines, which call their subroutines, so on. An example of that, from my programming experience, is when I wrote a program in the early 1980's to plot and display ‘milestone charts’ using a graphics package (Plot10 on the PDP/VAX) which basically only had 'move' or 'draw ' (from current point to the next point) routines. If I had not been able to use abstraction, to logically break my program into separate functional parts, I do not feel I would have finished it before the plug would have been pulled on the project.

I feel that sometime training can be given, which is hard to translate to the "real world" in which one is working. For example, I took a data structure class in the late 1970's. In which I learned the concepts of the data structures, however it was not until later after the end of the class, that I had time to figure out how to create and apply those data structures in the computer languages that I knew.

Modularity is perfectly designed for black box testing. When debugging you can check starting from where the error was discovered and work backward & upward. That is, check what values were passed into the subroutine, in which the error was discovered, and if those values are correct then the error is inside of the subroutine. If those values being passed into the subroutine are incorrect then an error will be found up at least one level (in the calling routines), and back through the sequential processing that used to get to that point in the program. If this is the case the program will then need to be re-tested after fixing the error found, to make sure that the subroutine does not have an error in the subroutine in addition to the one found upstream (i.e. pass the correct values and hunt for more errors).

Unless a programmer knows some of these techniques and is given time learn and program, then a COBOL program with no subroutines and global variables, seems (or is) easier to code and debug.

I know that at this stage in the system life cycle, the only time it could be used on legacy systems are would be in those parts of the system being repaired due to errors or have there requirements changed. Modular fixes/changes should be easier to reengineer into an OO environment, if re-engineering is done. Modularity may have never been a silver bullet, but it may be a silver pickle fork, useful nonetheless in conquering many programming problems.

The number of subroutines in the system per (divided by) the number of main programs in the system, and the number of subroutines each program in the system contains would be possible metrics that could be used to show degree of modularity found in the system.

And now changing the subject to metrics. In my opinion, the big picture view of metrics; is that metrics are the classification and measurement of objects, and events.

"Introduction to Mathematics for Life Scientists" (Third Edition) by Edward Batschelet gives four levels of classification/measurement: nominal, ordinal, interval, and ratio.

In a nominal scale, objects and events are named and defined. If you cannot give a name to something, that is to create a definition that makes it possible to decide whether the object or event is one (the named), then the subject needs more definition.

An ordinal scale is a nominal scale to which ranking (or ordering) has been added. The four generations of computer languages could either be considered a nominal ranking or be considered an ordinal ranking, depending on whether you feel that the languages are ranked lower too higher, or just named. Since the computer language generations are numbered, whoever invented this classification most likely considered it an ordinal scale.

An interval scale is an ordinal scale in which the intervals are meaningful (however the reference point is arbitrary). Temperature measured in Celsius is an example of an interval scale, so is time.

A ratio scale is an interval scale with an absolute reference point. Temperature measured in Kelvin is a ratio scale, which uses the same intervals as the Celsius, but does not have an arbitrary zero. Ratios and percentages are meaningful only in a ratio scale.

Each metric should be analyzed to determine to which classification/measurement scale it belongs. That would, then determine how the metric could be used, and whether percentages should be used.

Accounting can be thought of as science of tracking certain units (dollars for example) and in that sense, a science of metrics collection and reporting.

In one type of accounting the books are done to the cent (or other unit of measurement). Another type of accounting sets an acceptable error size (perhaps as percentage of some amount being tracked) before doing the books, then if the error is below that level then an adjustment to the books is made; otherwise errors are located until the remaining errors are below that limit.

An extreme example of accounting done to the penny can be found in the book, “The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage” by Clifford Stoll.

This true story tells how Clifford Stoll used a program to track system usage and when comparing it to the operating systems standard program used to track system usage, there were found errors of mere cents in the comparison. He then received permission from his boss to try to track down the difference (as a low priority workload and on his off time). The book goes on to show how he discovered and then tracked spying directed against the United States of America.

One may recall that in the 16th Century, the astronomer Tycho Brahe kept extraordinarily accurate observational data, which in the 17th Century was used by Johannes Kepler to prove that the earth and planets travel about the planets in elliptical orbits. If Tycho Brahe had not been a stickler for data accuracy and precision, then it may have taken many centuries to prove the Copernican theory.

One reason for the other style of accounting is to conserve money, why pay a professional "x" number of dollars per hour to try to find a 2 cent error. Another example, "In engineering problems, the data are seldom known to with an accuracy greater than 0.2 percent." Mechanics for Engineers -- STATICS and DYNAMICS by Ferdinand P. Beer and E. Russell Johnston, Jr., so on cannot get absolute accuracy unless the data has absolute accuracy. In engineering where it could be shown that the answer is not significantly affected then certain metrics might not need to be collected or used.

Any metric that is created by using mathematical operations on other metrics should use numerical analysis to maintain as much accuracy as possible, from it source metrics. See the following books: Elementary Numerical Analysis by W. Allen Smith, An Introduction to Error Analysis — The Study of Uncertainties in Physical Measurements by John R. Taylor.

In conclusion each metric should be analyzed to determine to which classification/measurement scale it belongs. Decisions need to be made as to the amount of effort to be made in collecting each metric, and in the accuracy at which it needs to be collected. Also I feel that we should borrow as many metrics and as much metrics theory as possible from other professional disciplines.